Guia de Desenvolvimento

Instruções de preenchimento do documento

Este documento deverá seguir as seguintes considerações de preenchimento:

- **Fonte:** Arial
- **Tamanho:** 10
- **Cor:** Preta
- **Sigla e nome do Projeto:** Caixa Alta
- **[...]:** Retirar os *colchetes* ao transcrever o texto
- Quando algum item não for aplicado a este documento informar a sigla **N/A**

[SIGLA] – [NOME DO PROJETO]

Gestor do Projeto
Klaymer Paz
klaymer.paz@saude.gov.br
3315-2212

Histórico de Revisão

Data	Versão	Nº Ordem de Serviço	Descrição	Autor
26/12/2018	1.0	04.2018	Criação do documento	Eduardo Nascimento

Sumário

1. INTRODUÇÃO

A proposta deste documento é definir a arquitetura básica e os conceitos fundamentais dos sistemas desenvolvidos pelo DATASUS. Estes padrões e diretrizes são apresentados em forma de visões arquiteturais que visam cobrir os principais aspectos técnicos relativos a estrutura e ao desenvolvimento dos sistemas.

2. OBJETIVO

Este documento tem por objetivo levantar de maneira mais detalhada o cenário da arquitetura atual dos sistemas existentes e a proposta/guia de desenvolvimento com Metodologia DevOps.

3. ARQUITETURA ATUAL

3.1 Estrutura das aplicações

Atualmente a arquitetura do DATASUS está estruturada de forma monolítica, onde é estabelecida uma aplicação de camada única no qual a interface do usuário e o código de acesso a dados são combinados em um único programa a partir de uma única plataforma.

Em linhas gerais, podemos dizer que é como um grande contêiner em que todos os componentes de software de uma aplicação são reunidos e compactados. Se pegarmos um caso específico de linguagem de programação Java, esse pacote pode ter vários formatos como EAR, WAR, JAR etc., que é finalmente implementado como uma única unidade no servidor de aplicações. Em termos de negócios, podemos dizer que: todos os diferentes serviços de negócios são agrupados como uma única unidade (fortemente acoplada).

3.2 Problemas identificados

A arquitetura monolítica do DATASUS apresenta complexidade de entendimento na totalidade das aplicações, decorrendo das seguintes dificuldades:

- Manutenibilidade complexa;
- Reutilização limitada;
- O dimensionamento das aplicações é um desafio muito considerável;
- É difícil obter agilidade operacional na implementação repetida de artefatos nas aplicações;

Por definição, essas aplicações foram implementadas usando uma única pilha de desenvolvimento, o que limita a disponibilidade de uso e escolhas de ferramentas, frameworks, tecnologias mais adequadas para determinada situação de trabalho.

3.3 Manutenção

Há um desafio grande e custoso para manter as aplicações atuais, pois, além de contar com uma variedade de codificação, a proporção de implementações feitas é tão elevada que é difícil de saber ao certo onde serão impactadas as alterações e as correções feitas nos sistemas.

As aplicações são muito grandes e complexas para entender e fazer alterações de maneira rápida e correta;

O tamanho das aplicações podem diminuir o tempo de inicialização;

É necessário reimplantar a aplicação inteira em cada atualização;

O impacto de uma mudança geralmente não é bem compreendido, o que leva a um extenso teste manual.

A implantação contínua é difícil;

As aplicações também podem ser difíceis de escalar quando diferentes módulos têm requisitos de recursos conflitantes;

Outro problema com as aplicações é a confiabilidade. Bug em qualquer módulo (por exemplo, vazamento de memória) pode potencialmente derrubar todo o processo. Além disso, como todas as instâncias da aplicação são idênticas, esse bug pode afetar a disponibilidade de toda a aplicação;

As aplicações têm uma barreira para adotar novas tecnologias. Como mudanças nos frameworks ou idiomas pode afetar a aplicação inteira, e isso se torna extremamente caro em termos de tempo e custo.

3.4 Impactos

Agilidade: O ciclo completo de desenvolvimento neste tipo de estrutura é custoso, pois as aplicações podem acomodar mudanças, mas ao custo de menor agilidade, porque mesmo se um pequeno componente em uma aplicação tiver que ser alterado, toda a aplicação precisar ser reempacotado e montado junto.

Escalabilidade: Em determinados momentos as aplicações atuais podem receber uma massiva de dados com vários usuários simultâneos, com isso o front-end ganha muita tração do lado do cliente e a necessidade surge quando a nossa aplicação não consegue mais lidar com novos acessos e nem com novos usuários, então nesse caso iremos precisar escalar nossa aplicação verticalmente e horizontalmente, para que possamos atender mais usuários.

É necessário criar mais instâncias da mesma aplicação para poder atender aos usuários finais sem prejudicar a performance, mas os recursos utilizados em outros serviços estão sendo desperdiçados, já que não precisam escalonar. Então podemos concluir que a escalabilidade também perde por que estamos escalando recursos para uma parte e não para o todo.

4. RISCOS

Os riscos de manter as aplicações do cenário atual são tão altos, pela falta de controle, monitoramento e outros fatores já citados acima que é provável que com o passar dos anos não será mais possível fazer manutenções nos softwares por não saber aonde vai impactar e o que vai impactar e possivelmente será optado por refazer tudo e congelar as versões atuais, por falta de suporte, ferramentas, bugs e as tais chamadas caixas pretas.

4.1 Arquitetura Proposta

Quando falamos de qual seria melhor arquitetura para o negócio, o ideal é que possamos olhar para onde o mercado está e a arquitetura para o cenário atual. As principais empresas de tecnologias do mercado mundial estão indo em relação a inovação e desenvolvimento de software e as metodologias que elas estão usando para que a qualidade de seus produtos sejam cada vez mais confiável, rápidos e usual. O Microserviços é a atual arquitetura em conjunto com o ciclo de DevOps e é para onde o mercado está indo.

4.1 Microserviços

Grandes players e vários movimentos da indústria recente estão promovendo o foco e, em seguida, a adoção deste importante conceito em uma grande variedade de soluções de aplicações de software.

Microserviços abrangem automação e uma vez que tenhamos uma pipeline plausível para empacotar e entregar seus Microserviços (Entrega contínua e integração contínua) com qualquer suíte de automação que queira ou tenha hoje, devemos ser capazes de obter as melhores vantagens dessa abordagem desde o início, desde os primeiros Microserviços Produção.

A automação é fundamental para expandir no futuro o número de Microserviços na solução. A maneira normal de fazer isso é depois de começar a decompor o monolítico atual. Ele deve suportar a evolução herdada com a adoção desse estilo arquitetônico.

É fundamental também porque precisamos garantir o nível de qualidade de entrega no produto final. Com a automação e controle adequados, podemos alcançar o resultado desejado de maneira mais rápida.

Os microserviços podem ter equipes dedicadas para dois microserviços diferentes que precisam se comunicar uns com os outros. A mentalidade da equipe em torno desse contexto comercial específico (conceito Contexto de Limite/Design Dirigido por Domínio) deve evoluir naturalmente devido aos desafios e demandas de negócios diários que ocorrem.

É normal que sua especialidade cresça cada vez mais, dando a necessária liberdade de escolha para a equipe, para que ela possa definir a melhor maneira de fornecer o serviço comercial necessário. A equipe adotar as mudanças e as entrega de acordo com o esperado.

Como facilidade, temos que uma vez que tenha equipes específicas para evoluir e administrar os microserviços e a base de código segmentada por microserviços, as alterações mínimas que colocarmos no código não diminuirão toda a solução, devido a referência cruzada em tempo de compilação ou tempo de execução. Não precisamos reconstruir a solução inteira porque editou uma única dependência realmente usada em outro módulo. Isso é passado. O teste também deve ser mais fácil (até certo ponto) quando tivermos o escopo de negócios específico para testar e validar.

Os microserviços são executados em unidades de execução separadas, chamados contêineres. Os contêineres fornecem todos os recursos necessários pelos serviços. Para cada microserviços, devemos ter um contêiner dedicado que ofereça suporte a ele. É por isso que a automação, nesse caso, é tão importante (não apenas para empacotar e testar), mas também para entregar o produto final, com todos os recursos que são necessários para o serviço, entregues no formato de um contêiner com o microserviço embutido.

Depois de entregar o contêiner, podemos dimensioná-lo de forma independente, cada contêiner com seu único nível de escalabilidade, de acordo com as necessidades da solução como um todo.

A implementação de microserviços é flexível. Cada microserviço pode ser escrito em uma linguagem de programação diferente, apenas porque eles são executados separadamente e se comunicam de forma eficaz através de interfaces bem definidas sobre protocolos conhecidos, como JSON. Isso dá flexibilidade a solução, uma vez que podemos explorar as melhores abordagens e recursos de cada linguagem de programação.

Os microserviços devem abordar um domínio de negócios específico e oferecer um ótimo valor comercial por meio de sua documentação bem definida e pública e da API, como, por exemplo, usar um mecanismo de documentação de API's como o swagger.

Em essência, esses são os principais motivadores que acreditamos que devemos considerar quando começar a pensar em um novo design de solução usando o estilo de arquitetura de microserviços (MSA).

A base da arquitetura de micros serviço (MSA) é sobre o desenvolvimento de uma única aplicação como um conjunto de serviços pequenos e independentes que estão sendo executados em seus próprios processos, desenvolvidos e implantados de forma independente, ou seja, um microserviço tem que ser autossuficiente.

A maioria das definições de MSA explica isso como um conceito arquitetônico focado em segregar os serviços disponíveis em um conjunto de serviços independentes. No entanto, os microserviços não estão apenas dividindo os serviços disponíveis em serviços independentes;

Considere que, olhando para as funções oferecidas pelo serviço, identificando os recursos de negócios exigidos da aplicação - ou seja, o que a aplicação precisa fazer, para ser útil. Em seguida, esses recursos de negócios podem ser implementados como serviços (independentes) totalmente independentes, refinados e autônomos (micro). Eles podem ser implementados em cima de diferentes pilhas de tecnologia, mas, no entanto, cada serviço estaria abordando um escopo de negócios muito específico e limitado. Dessa forma, o cenário de um sistema on-line que apresentamos acima pode ser realizado com um MSA. Como podemos ver, com base nos requisitos de negócios, há um microserviço adicional criado a partir do conjunto original de serviços. É evidente, então, que isso vai além de dividir os serviços e chegar a terrenos mais complexos.

4.1.1 Projetando Microserviços

Com o objetivo de converter nossas aplicações e serviços existentes em microserviços, é importante que possamos decidir adequadamente o tamanho, o escopo e os recursos dos

microserviços. Esta é talvez e a coisa mais difícil que possamos encontrar inicialmente quando formos implementar o MSA na prática.

Algumas das principais preocupações práticas e equívocos sobre o assunto:

Linhas de código/tamanho de equipe são métricas ruins: Existem várias discussões sobre como decidir o tamanho dos microserviços com base no número de linhas de código da implementação ou no tamanho de sua equipe (ou seja, equipe de duas pizzas). No entanto, essas são consideradas métricas muito impraticáveis e ruins, porque ainda podemos desenvolver serviços que violam completamente os princípios de arquitetura de microserviços com menos código e equipes de duas pizzas.

'Micro' é um termo um pouco enganador: a maioria dos desenvolvedores tende a pensar que eles deveriam tentar tornar o serviço o menor possível. Isso é uma má interpretação.

No contexto de serviços da Web/SOA, os serviços geralmente são implementados em diferentes granularidades - de algumas funções a várias dezenas de funções. Ter webservices e reformulá-los como microserviços não lhe trará nenhum benefício de MSA.

4.1.2 Mensagens em Microserviços

Em aplicações monolíticas, as funções de negócios de diferentes processadores/componentes são chamadas usando chamadas de função ou chamadas de método em nível de idioma. No SOA, isso foi transferido para um sistema de mensagens de nível de serviço da Web muito mais fracamente acoplado, que é baseado principalmente no SOAP, além de protocolos diferentes, como HTTP, JMS.

4.1.3 Gerenciamento de Dados

Na arquitetura monolítica, a aplicação armazena dados em bancos de dados únicos e centralizados para implementar várias funções/capacidades da aplicação. No MSA, as funções estão dispersas em vários microserviços e, se usarmos o mesmo banco de dados centralizado, é difícil garantir o baixo acoplamento entre os serviços (por exemplo, se o esquema do banco de dados tiver mudado de um determinado microserviço, isso quebra vários outros serviços). Portanto, cada microserviço precisaria ter seu próprio banco de dados.

Cada microserviço pode ter um banco de dados privado para persistir os dados necessários para implementar a funcionalidade comercial oferecida a partir dele. Um determinado microserviço só pode acessar o banco de dados privado dedicado, mas não os bancos de dados de outros microserviços. Em alguns cenários de negócios, talvez seja necessário atualizar vários bancos de dados para uma única transação. Em tais cenários, os bancos de dados de outros microserviços

devem ser atualizados apenas por meio de sua API de serviço (não é permitido acessar diretamente o banco de dados).

O gerenciamento de dados descentralizados oferece microserviços totalmente desacoplados e a liberdade de escolher técnicas de gerenciamento de dados (SQL ou NoSQL etc., sistemas de gerenciamento de banco de dados diferentes para cada serviço). No entanto, para casos de uso transacionais complexos que envolvem vários microserviços, o comportamento transacional deve ser implementado usando as APIs oferecidas de cada serviço e a lógica reside no nível do cliente ou intermediário.

4.1.4 Registro de Serviços

No MSA, o número de microserviços com os quais vamos precisar lidar é bastante alto. Suas localizações mudam dinamicamente devido à natureza rápida e ágil de desenvolvimento/implantação de microserviços. Portanto, precisaremos encontrar a localização de um microserviço durante o tempo de execução. A solução para esse problema é e usar um registro de serviço.

O registro de serviço contém os metadados das instâncias de microserviço (que incluem seus locais reais, porta do host, etc.). As instâncias de microserviço são registradas no registro de serviço na inicialização e cancelado o registro no desligamento. Os consumidores podem encontrar os microserviços disponíveis e suas localizações através do registro de serviços.

4.1.5 Descoberta de Serviço

Para encontrar os microserviços disponíveis e sua localização, precisamos ter um mecanismo de descoberta de serviço. Existem dois tipos de mecanismos de descoberta de serviço - descoberta no lado do cliente e descoberta no lado do servidor. Vamos dar uma olhada mais de perto nesses mecanismos de descoberta de serviço:

4.1.6 Descoberta do lado do cliente

Nessa abordagem, o cliente ou o API Gateway obtém o local de uma instância de serviço consultando um registro de serviço.

4.1.7 Implantação

O Docker (um mecanismo de software livre que permite que desenvolvedores e administradores de sistemas implantem contêineres de aplicações autossuficientes em ambientes

Linux) fornece uma ótima maneira de implantar microserviços que atendam aos requisitos acima. Os principais passos são:

- Empacotar o microserviço como uma imagem de container (Docker).
- Implementar cada instância de serviço como um contêiner.

O dimensionamento é feito com base na alteração do número de instâncias do contêiner.

Construir, implantar e iniciar um microserviço será muito mais rápido, já que estamos usando contêineres do Docker (que é muito mais rápido do que uma VM comum).

O Kubernetes está estendendo os recursos do Docker, permitindo gerenciar um cluster de contêineres Linux como um sistema único, gerenciando e executando containers Docker em vários hosts, oferecendo colocation de contêineres, descoberta de serviço e controle de replicação.

Logo, a maioria desses recursos também é essencial no contexto de microserviços. O uso do Kubernetes (no topo do Docker) para implantação de microserviços é uma abordagem extremamente poderosa, especialmente para implantações de microserviços em larga escala.

4.1.8 Segurança

A proteção de microserviços é um requisito bastante comum quando usamos microserviços em cenários reais. Antes de entrar na segurança de microserviços, vamos dar uma rápida olhada em como normalmente implementamos a segurança no nível de aplicação monolítica.

Em uma aplicação monolítica típica, a segurança diz respeito a descobrir que "quem é o chamador", "o que o chamador pode fazer" e "como propagamos essa informação".

Isso geralmente é implementado em um componente de segurança comum, que está no início da cadeia de manipulação de solicitações, e esse componente preenche as informações necessárias com o uso de um repositório de usuários subjacente (ou repositório de usuários).

Isso requer um componente de segurança implementado em cada nível de microserviços que esteja falando com um repositório de usuários centralizado compartilhado e recupere as informações necessárias. Essa é uma abordagem muito tediosa para resolver o problema de segurança de microserviços.

Em vez disso, podemos aproveitar os padrões de segurança de API amplamente usados, como OAuth 2.0, para encontrar uma solução melhor para o problema de segurança de microserviços. Antes de mergulhar nisso, vamos primeiro resumir o propósito de cada padrão e como podemos usá-los.

OAuth 2.0 - é um protocolo de delegação de acesso. O cliente autentica-se com um servidor de autorização e obtém um token opaco, conhecido como 'token de acesso'. O token de acesso tem zero informações sobre o usuário/cliente. Só tem uma referência as informações do usuário que só

podem ser recuperadas pelo servidor de autorização. Portanto, isso é conhecido como um 'token de referência e' é seguro usar esse token mesmo na rede pública/internet.

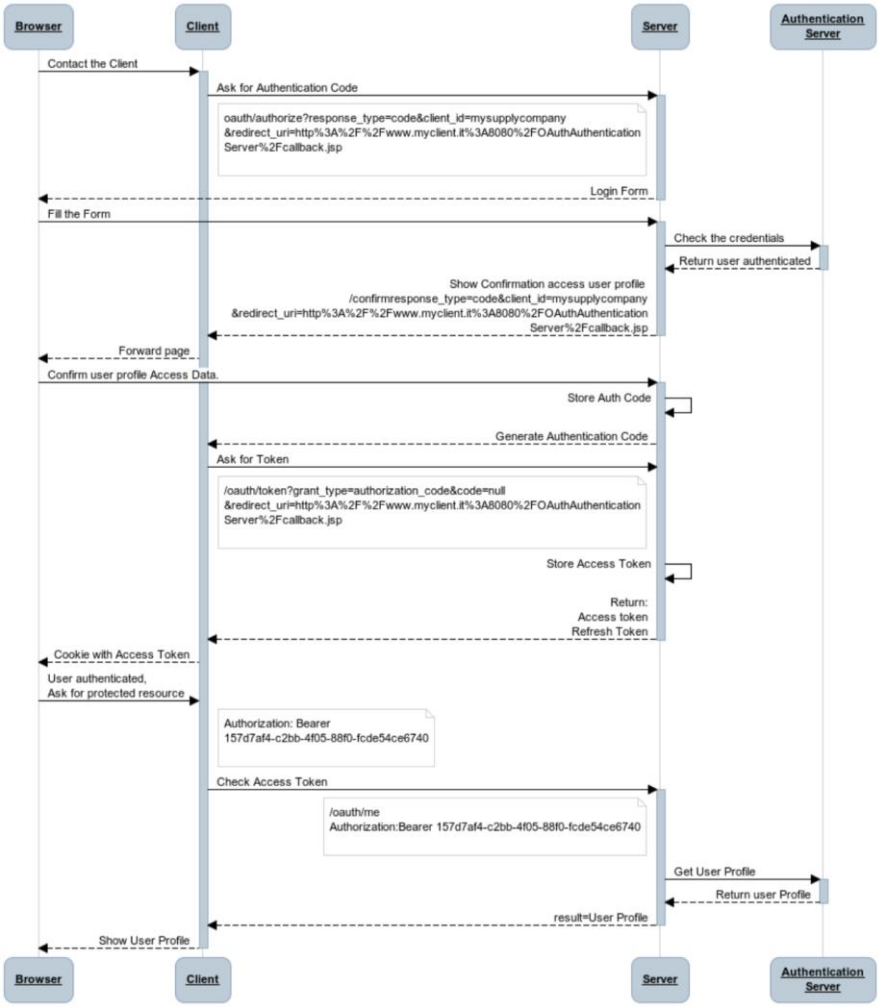

Figure 5: Fluxo Oauth

4.2 Separação do front-end do back-end

Figure 6: Separação do front-end do back-end

A separação de front-end do back-end traz ganhos e benefícios.

4.2.1 Escalabilidade

É mais fácil dimensionar uma aplicação monolítica desacoplando o client do server. Primeiro como o código é dividido em duas partes, nesse caso podemos otimizar o código mais rapidamente. Em segundo lugar, podemos aumentar os recursos para o front-end e o back-end em um ritmo separado, já que o back-end precisaria ser incrementado em um ritmo relativamente mais rápido conforme ele cresce. Com essa forma, é possível escalonar de maneira mais adequada o que mais precisa de recurso. Sendo que normalmente frameworks responsivos na demanda tanto custo de processamento e podemos colocá-los em equipamentos de menores recursos e servidores HTTPs mais leves como apache ou nginx;

4.2.2 Otimização de Recursos

Nesse cenário os servidores fazem todo o trabalho, analisando solicitações, recuperando dados relevantes de um banco de dados, aplicando cálculos ou alterações nos dados com base no que é solicitado e, em seguida, colocando tudo junto em formato HTML usando um modelo e enviá-lo ao navegador. Considerando que, o lado do cliente, os servidores http apenas enviam dados brutos principalmente no formato JSON - para o navegador, o que os une de maneira perfeita e rápida. Os servidores precisam trabalhar menos e a experiência do usuário é melhor, pois a página inteira não está sendo atualizada em todas as solicitações. Vantajoso para as duas partes.

4.2.3 Atualização mais fácil

Atualizando seu quadro pode ser uma dor real (pelo custo de mudança de paradigma de desenvolvimento a curto prazo). Mas manter o back-end e o front-end separados diminuirá suas chances de quebrar o sistema inteiro. É mais fácil depurar, pois já saberíamos se é um problema de front-end ou de back-end;

4.2.4 Mais simples para alternar estruturas

Quando estamos falando de tecnologia, devemos ter em mente que ela muda rapidamente. Precisamos acompanhar as últimas inovações para se manter competitivo. Quanto menor a percentagem do modo, mais fácil é trocá-lo. No nosso caso, digamos que através de testes e conclusão decidimos trocar a linguagem de programação de nossos sistemas por que o mesmo teoricamente irá funcionar mais rapidamente com o Python, nesse caso precisaremos alterar completamente o nosso código front-end, mesmo que queira manter toda a aparência. Enquanto que se desacoplarmos um do outro, digamos que os nossos testes indiquem que os desenvolvedores do Facebook e a comunidade de software livre fizeram um ótimo trabalho com o ReactJS e funcionem

melhor em nosso caso particular, nesse caso precisaríamos mexer no código de back-end para mudar nosso framework responsivo web para React;

4.2.5 Implantação mais rápida

Como o trabalho de um desenvolvedor de front-end não depende muito do trabalho do desenvolvedor de back-end e vice-versa, seu código pode ser testado e potencialmente implantado sempre que for feito sem esperar pela conclusão da outra pessoa, pelo menos para projetos que estão apenas desenvolvendo ou otimizando as APIs antigas.

4.2.6 Consolidação de API's

Neste mundo cada vez maior de dispositivos, temos mais e mais versões de código para gerenciar (aplicação web, aplicativo iOS, aplicativo Android, etc.). Na maioria dos casos, estamos fornecendo as mesmas informações para todos essas aplicações. Portanto, só faz sentido atendê-los a partir da mesma base de código. Ter uma aplicação baseada em API torna a vida dos desenvolvedores mais simples. Ter menos código para lidar com isso torna mais gerenciável.

4.2.7 Modularidade

Se o front-end e separado do back-end, torna-se mais fácil trabalhar em um módulo mantendo o outro intocado. A modularidade também torna mais fácil para duas equipes ou duas pessoas trabalharem no front-end e no back-end simultaneamente sem se preocupar em sobrescrever ou atrapalhar o trabalho de outras pessoas. Eles podem trabalhar em um ritmo diferente também.

4.3 DevOps

A implementação de DevOps e Entrega Contínua para o cenário atual objetiva tirar o máximo proveito de um switch para DevOps e requer novas ferramentas, novos processos e uma mudança cultural significativa.

Desde a sua criação, a adoção de DevOps foi objeto de debates nos círculos de TI. Alguns consideraram moda; outros acreditaram, desde cedo, que revolucionaria as operações de TI. Ano após ano, os analistas preveem um crescimento rápido. Mas o fato é que o DevOps é um movimento que cresce a um ritmo constante, e ainda não passou a ser regra em muitas organizações.

O DevOps evoluiu a partir de uma metodologia para reunir as equipes de desenvolvedores e da área de operações ao redor de uma estratégia para transformar o negócio, com rapidez e segurança. Ao facilitar uma maior comunicação, colaboração e integração em toda a organização, o

DevOps melhora significativamente a agilidade da prestação de serviços de TI e simplifica o gerenciamento de TI l, ao mesmo tempo em que otimiza custos.

Na prática, DevOps é uma filosofia e prática que reúne equipes de desenvolvimento, operações e testes em equipes multifuncionais, cada uma das quais responsável por todo o ciclo de vida de um produto ou serviço. No caso da TI, produtos ou serviços de software. Ao reunir equipes colaborativas em toda a organização, o DevOps cria um ambiente operacional estável para lançar o código mais rápido no mercado, reduzindo erros sistêmicos e humanos, aumentando o controle das versões, otimizando os custos e recursos.

A jornada DevOps começa explorando os procedimentos comerciais atuais e os pipelines de entrega, e identificando objetivos claros que a instituição deseja que sua estratégia de desenvolvimento alcance. É necessário decidir se enfrenta a implantação Green eld ou Brown eld. As implantações do Green eld são criadas a partir do zero, tornando-as mais fáceis de implementar, mas propensas ao tempo de inatividade a medida que os serviços são provisionados através da nova metodologia de implantação. As implementações Brown eld resultam em menos tempo de inatividade, mas muitas vezes são mais difíceis de implementar devido a necessidade de confiar em metodologias de implantação paralelas.

Devops integra pessoas, processos e ferramentas em um conjunto voltado para transformar uma organização em uma única entidade. Como tal, a mudança cultural e a espinha dorsal do modelo, e mudar a cultura da sua empresa será um dos desafios mais difíceis a ser enfrentado. Introduzir novas políticas e procedimentos, embora bem-sucedidos, inicialmente, poderão falhar no longo prazo, a menos que a cultura organizacional subjacente também seja alterada.

DevOps é um movimento de toda a empresa, desde os executivos C-level. Será necessário dissociar o nome da função e garantir que os desenvolvedores e o pessoal de operações estejam informados sobre o valor que cada um traz para a organização antes de reuni-los em equipes multifuncionais.

Para facilitar a mudança cultural, os incentivos também devem mudar. O modelo de incentivo mais eficaz é recompensar as equipes multifuncionais para oferecer uma melhor experiência ao cliente, ao mesmo tempo em que faz com que as falhas sejam tão baratas quanto possível. Algumas organizações pedem aos desenvolvedores que trabalhem on-call para que eles possam entender os desafios das operações. Alguns aceleram a mudança cultural ao identificar os desenvolvedores de rock-star e equipes de operações para ajudar a motivar outros membros da equipe. Outros ajudam a preparar um líder agradável para cada equipe para ajudar a tornar a transição para o DevOps muito mais suave.

4.4 Integração Contínua

Para criar uma integração contínua e uma plataforma de entrega contínua, seu foco deve se voltar para a própria equipe DevOps. A principal questão é fornecer aos desenvolvedores, informações precisas e atualizadas sobre o ambiente de produção para que possam planejar a implantação adequadamente.

Isso garantirá que os desenvolvedores possam se concentrar em uma abordagem coordenada de "construir e executar", na qual o desenvolvedor que constrói um produto ou serviço, ele "possui" esse produto ou serviço até a produção. Esta "propriedade" pode prolongar-se por um período de tempo definido, para garantir que os principais erros sejam abordados antes da entrega, por toda a vida útil do produto ou serviço. Ambas as abordagens funcionam.

Figure 8: Integração Contínua

A equipe do DevOps toca todos os pontos do ciclo de vida do serviço, desde os requisitos até o planejamento, implantação e manutenção. Esta equipe, com seu profundo conhecimento da plataforma e da infraestrutura, também soluciona problemas - uma função que tradicionalmente recai sobre o pessoal de operações. Aqui, a chave é criar um pipeline de implantação automático, em que a implantação de um script totalmente automatizado em qualquer ambiente ocorre em poucos minutos. O pipeline de implantação deve integrar integração contínua, desenvolvimento contínuo, testes contínuos e implantação contínua em uma única entidade.

4.4.1 Fluxo

Uma mudança de código comprometida com um sistema de controle de origem desencadeia um processo em um servidor de compilação.

Este processo compila o código, reunindo os artefatos necessários para a implantação de clientes em pacotes implementáveis.

Testes automatizados são realizados nesses pacotes para garantir que a qualidade do código seja mantida e o código seja executado conforme o esperado.

Se o estágio de confirmação for bem-sucedido, o aplicativo é implantado em um ambiente de teste onde pode ser inspecionado por uma equipe de garantia de qualidade, por exemplo.

Finalmente, uma vez que o aplicativo foi aprovado, essa mesma versão pode ir direto para a produção.

4.5 Ambiente de teste contínuo

Quanto mais rápido receber feedback sobre as mudanças, melhor será a qualidade do software. No processo tradicional, o código completo passa do desenvolvimento para a área de teste e é então empurrado para a produção, caso aprovado. Caso contrário, o código será enviado de volta ao desenvolvimento para edição. Isso leva mais tempo e é menos confiável.

No modelo DevOps, o teste torna-se parte integrante do desenvolvimento, e o pessoal de QA e parte da equipe de transposições multifuncionais. O teste, manual ou automatizado, é realizado continuamente em toda a tubulação de entrega. Toda mudança é tratada como um potencial candidato a liberação, e o objetivo é manter o tempo mais curto possível entre o check-in e a liberação.

Como medida de segurança, uma entrega de esqueleto com um teste de unidade única e um teste de aceitação único que esteja integrado ao seu script de implantação automatizada devem ser criados. A medida que avançar, pode-se aumentar o número de testes e espalhá-los em toda a tubulação de entrega.

4.6 Implementação de Integração Contínua

A implementação contínua estende a entrega contínua. Cada compilação que passa com sucesso por um ciclo de teste completo é implantada automaticamente. Elimina a necessidade de intervenção humana para decidir o que implementar e quando implementar. Com a implementação contínua, as organizações podem rapidamente fornecer novos recursos e atualizações, ao mesmo tempo em que proativamente modificam o produto.

Como não há verificação manual antes que o código seja implementado, a implementação contínua pode parecer um pouco assustadora. Você não precisa se preocupar em perder o controle sobre o código em produção quando você possui uma tubulação de entrega bem projetada. Além disso, o DevOps pode dar um controle mais detalhado sobre a funcionalidade de um determinado serviço, tornando certas compilações disponíveis apenas para usuários selecionados ou automatizando a liberação de recursos em um momento especificado.

Seus desenvolvedores devem garantir que o código que eles escrevem esteja bem desenhado, e o controle de qualidade deve criar conjuntos de testes que estão sempre atualizados. Quando o seu pipeline de entrega é projetado corretamente, o sistema decide automaticamente o que e quando reter, e o que e quando liberar para a produção e assim por diante.

A implementação contínua deve ser aumentada através de monitoramento contínuo e feedback. Com feedback inicial, os desenvolvedores sabem quais recursos são úteis para usuários finais. Isso os ajuda a concentrar-se nos recursos mais importantes, economizando tempo e esforço.

4.7 Alta disponibilidade

A alta disponibilidade é um recurso significativo da arquitetura corporativa. Arquitetura corporativa é uma necessidade para sistemas de alto perfil e ocupados. Especialmente para aqueles que precisam lidar com um grande número de acessos simultâneos e grandes quantidades de tráfego. Portanto, nem todos os sistemas e empresas exigem Arquitetura Empresarial.

Existem muitas razões pelas quais a Alta Disponibilidade é crucial para o DATASUS, mas podemos resumir estas razões das seguintes maneiras:

Confiabilidade: com a capacidade de ter um componente operando mesmo quando outros falham a alta disponibilidade permite uma operação contínua no sistema.

Escalabilidade: A maneira como os clusters são projetados aumenta a flexibilidade da infraestrutura - permite uma escalabilidade perfeita sempre que necessário. Nós até podemos escalar em dois níveis diferentes. A escala horizontal garante que nós extras sejam adicionados quando o tráfego aumentar. O escalonamento vertical, por outro lado, pode implicar a adição de mais memória ou uma atualização da CPU, dependendo da ocasião e das necessidades de um sistema/empresa.

Manutenção melhor: quando se tem um negócio on-line, sabe que não se pode arriscar a possibilidade de sistema ser quebrado ou lento ou, na pior das hipóteses, sofrer um tempo de inatividade. Ao mesmo tempo, temos que estar cientes de que quando se está on-line precisamos dar manutenção contínua e meticulosa. O uso de uma arquitetura corporativa de várias camadas com alta disponibilidade integrada permite procedimentos de manutenção melhores e contínuos. Atualizar o Software enquanto estiver em uma Arquitetura Corporativa é um procedimento menos propenso a erros, já que o uso de clusters ativo-ativo mitiga o risco de algo ficar extremamente errado. Ao executar os procedimentos de manutenção, cada nó é retirado do conjunto de nós ativos, atualizado, mantido e verificado e retornado ao conjunto onde, durante todo o tempo, os nós remanescentes continuam a funcionar sem problemas.

Melhor segurança: são novamente os níveis separados de aplicações e funcionalidades que concedem um grau diferente e até mesmo um tipo diferente de segurança a ser aplicado em cada camada. Por exemplo, vários firewalls podem ser encontrados entre cada camada ou o acesso a camadas e segmentado dependendo do usuário e sua função. Desta forma, a segurança também é construída e formatada de forma a permitir o máximo grau de segurança.

Os benefícios da Alta Disponibilidade são muitos e não podem ser ignorados, especialmente quando tratamos de um negócio altamente complexo e on-line que são vibrantes e tende estar sempre em expansão.

Desfrutando de menos tempo de inatividade isso tem uma série de vantagens, desde proporcionar uma melhor experiência do usuário e navegabilidade até obter mais receita e estabelecer uma presença online confiável.

Nesse caso o órgão apresenta ser mais confiável para seus usuários e sua credibilidade aumenta, assim como a fidelidade dos clientes e usuários.

Economicamente, menos receita é desperdiçada, pois há menos interrupções. Isso é particularmente importante quando tratamos de um sistema on-line, se o mantermos sempre funcionando corretamente com as provisões finais de segurança. Cada minuto o sistema pode experimentar lentidão ou tempo de inatividade e com isso causar glosas.

A produtividade é aprimorada e as horas que usuários, colaboradores, clientes e funcionários desperdiçam ao tentar navegar e trabalhar em um sistema que não oferece condições de trabalho on-line rápidas, seguras e confiáveis tem um alto custo.

4.8 Qualidade do Código

Garantir a qualidade do código quando a equipe de software está crescendo rapidamente é um grande desafio. Mas mesmo com um número constante de desenvolvedores de software, não ter a qualidade do código pode causar dores de cabeça.

Sem ferramentas e um sistema consistente, todo o projeto pode acumular uma enorme dúvida técnica, causando mais problemas a longo prazo do que resolve a curto prazo.

A qualidade do código é um grupo de atributos e requisitos diferentes, determinados e priorizados pelo seu negócio. Os principais atributos que podem ser usados para determiná-lo:

Clareza: Fácil de ler e supervisionar para quem não são o criador do código. Se for fácil de entender, é muito mais fácil manter e estender o código. Não apenas computadores, mas também os humanos precisam entender isso;

Sustentável: um código de alta qualidade não é complicado demais. Qualquer pessoa que trabalhe com o código precisa entender todo o contexto do mesmo, se quiser fazer alguma alteração;

Documentado: A melhor coisa é quando o código é autoexplicativo, mas é sempre recomendável adicionar comentários ao código para explicar seu papel e funções. Isso torna muito mais fácil para qualquer pessoa que não tenha participado da redação do código para compreendê-lo e mantê-lo;

Refatorada: A formatação de código precisa ser consistente e seguir as convenções de codificação do idioma;

Bem testado: Quanto menos bugs o código tiver, maior sua qualidade. Testes exaustivos filtram bugs críticos, garantindo que o software funcione da maneira pretendida;

Extensível: O código que devemos receber deve ser extensível. Não é muito bom quando temos que jogá-lo fora depois de algumas semanas.

E ciência: O código de alta qualidade não usa recursos desnecessários para executar uma ação desejada.

Um código de qualidade não atende necessariamente a todos os atributos mencionados acima, mas quanto mais ele atende, maior é sua qualidade. Esses requisitos são mais como uma lista de prioridades que depende das características do projeto.

5. INFRAESTRUTURA

5.1 Centralização de Logs

Ação: Implantar ambiente centralizado com os logs de aplicações e de acesso, com identificação das instâncias, servidores e demais informações para consulta.

Objetivo: Com os logs centralizados fica mais fácil identificar e controlar os erros das aplicações, podendo ser extraídas até informações de balanceamento de carga entre instâncias.

Ferramenta sugerida: GrayLog

5.2 Gerenciamento de API

Ação: Implantar gerenciamento das APIs

Objetivo: A principal ideia por trás do estilo de gateway de API é usar um gateway de mensagens leve como o principal ponto de entrada para todos os clientes/consumidores e implementar os requisitos não funcionais comuns no nível do gateway. Em geral, um gateway de API permite que consumamos uma API gerenciada em REST/HTTP. Como resultado, podemos expor nossas funções de negócios que serão implementadas como microserviços por meio do API gateway, como APIs gerenciadas. Na verdade, essa é uma combinação de gerenciamento de API e MSA.

Ferramenta sugerida: EUREKA + ZUUL

5.3 Code Quality

Ação: Implantar ambiente integrado entre códigos e ferramenta de análise de códigos.

Objetivo: Manter a qualidade do software dentro da meta estabelecida e só permitir se o código atingir o valor, garantindo uma maior qualidade para manutenção do software e para execução (evitar loops in nitos ou outros erros já conhecidos)

Ferramenta sugerida: Gitlab, sonar, Jenkins

5.4 Gerenciamento de Repositório de Artefatos

Ação: Atualizar ferramenta de gerenciamento de repositório.

Objetivo: Repositório de artefatos gerados pelo Apache Maven. A versão atual é a versão 2 do nexus. A atualização para a versão 3 não causa impactos e mantém uma ferramenta e tecnologia que já está em uso pela instituição. A migração para a versão 3 do Nexus irá suportar os repositórios com javascript e outros tipos de arquivos.

Ferramenta sugerida: nexus 3

5.5 Versionamento

Ação: Atualizar ferramenta de gerenciamento de versão.

Objetivo: O GitLab é um gerenciador de repositório de software baseado em git, com suporte a Wiki, gerenciamento de tarefas e CI/CD. GitLab é similar ao GitHub, mas o GitLab permite que os desenvolvedores armazenem o código em seus próprios servidores, ao invés de servidores de terceiros. Ele também oferece um guia de estilo de ramificação chamado GitFlow, que permite a colaboração perfeita entre os membros da equipe e facilita o escalonamento da equipe de desenvolvimento.

Fornece um sistema fácil de rastrear que separa o produto ativo do ramo de desenvolvedor menos estável com recursos não publicados. Quando um desenvolvedor finaliza um recurso, ele envia uma solicitação de recebimento no GitHub. Isso descreve o conteúdo e os detalhes da solicitação. Este método garante que nenhum código não revisado será mesclado com o ramo mestre.

Ferramenta sugerida: Gitlab

5.6 Integração Contínua

Ação: Implantar ambiente com integração contínua, desde o commit do desenvolvedor até o deploy da aplicação.

Objetivo: Documentar, versionar e medir a qualidade do código, evitando que códigos mal elaborados ou codificados sejam incluídos na aplicação e possa causar erros.

Ferramenta sugerida: Gitlab, sonar, Jenkins

5.7 Infraestrutura elástica

Ação: Implantar infraestrutura elástica, com containers e stacks para auxílio a cultura DevOps.e TI BI Modal.

Objetivo: Com a infraestrutura elástica, as aplicações podem ser controladas e facilmente monitoradas através da centralização de logs e painéis de monitoração. Além de que, o próprio desenvolvedor pode gerar um deploy de homologação ou desenvolvimento para testar o código. Com a integração contínua, o processo fica automático, sendo testado no ambiente, mas de forma isolada.

Auxilia também em caso de sobrecargas ou erros que possam acontecer, tratando com processos de auto scaling e self healing. Os processos de auto scaling consegue iniciar novas instâncias no caso de alta carga no sistema, reduzindo o impacto para o usuário no caso de acessos simultâneos foi além do esperado.

O processo de self healing executa a criação de novas instâncias quando identificados erros. No caso de uma reciclagem de instância (restart do servidor de aplicação, a própria infra destrói o container, cria um novo e direciona a chamada para o novo container, reduzindo assim os erros retornar amos(?) para o usuário e aumentando o nível de satisfação. Com a infraestrutura elástica também será possível aproveitar melhor os recursos computacionais da instituição.

A publicação em ambiente final de desenvolvimento pode ser configurada para ser realizada com a aprovação da infraestrutura. Todo o stack de produção pode ser configurado e aprovado pela infraestrutura.

Ferramenta sugerida: Docker, Openshift, kubernets, swarm, open stack, Amazon, Azure, Estaleiro SERPRO

5.8 Documentação de API

Ação: Implantar ferramenta de documentação de API.

Objetivo: Documentar as apis e permitir testes das mesmas.

Ferramenta sugerida: Swagger

6. REFERÊNCIAS

- Patterns of Enterprise Application Architecture, Martin Fowler et al., http://www.martinfowler.com
- Repository, JavaBuilding, URL: http://www.javabuilding.com/academy/patterns/repository.html
- REST API Design RuleBook, Mark Masse, http://shop.oreilly.com/product/0636920021575.do OWASP Top 10 { 2013 - https://www.owasp.org/images/9/9c/OWASPT op102013P TBR:pdf